REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	01 DEC 96	FINAL 06 Jun 92 through 15 Jun 96

4. TITLE AND SUBTITLE
Compiler Reorganization of Shared Data
(Subtitle: Compiler Technology)

5. FUNDING NUMBERS
N00014-92-J-1395
mod P00005

6. AUTHOR(S)
Professor Susan J. Eggers

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Washington
Department of Computer Science and Engineering
Box 352350
Seattle, WA 98195-2350

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Dr. Andre vanTilborg
800 N. Quincy Street
Arlington, VA 22217-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

False sharing in shared-memory multiprocessors is caused by a mismatch between the layout of write-shared data in memory and the cross-processor memory reference pattern to it. This mismatch can be eliminated by restructuring the data to match the pattern. We have developed compiler algorithms that do this: they analyze per-process shared data accesses in coarse-grained, explicitly parallel programs, pinpoint the data structures that are susceptible to false sharing, and choose an appropriate transformation to reduce it. The restructured programs execute up to three times faster on a KSR-2 multiprocessor, and have considerably better scalability (better performance with increasing numbers of processors). When compared to programmer efforts to restructure shared data (including the extensively hand-tuned SPLASH and SPLASH2 benchmarks), the compiler-directed approach never did worse, and for most programs did considerably better.

14. SUBJECT TERMS			15. NUMBER OF PAGES
false sharing	cache coherency	multiflow	11
compilers	multithreading		16. PRICE CODE: 0

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

ONR Three-year Review
Grant N00014-92-J-1395

Susan J. Eggers
Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195-2350
(206) 543-2118; fax: (206) 543-2969
eggers@cs.washington.edu

October 28, 1996

ONR Grant N00014-92-J-1395 funded one research project, Compiler Reorganization of Shared Data, in entirety and provided seed funding for another, Simultaneous Multithreading.

Compiler Reorganization of Shared Data

1 Technical Objectives

The increasing discrepancy between processor and memory speeds has made reducing the number of memory accesses tantamount to obtaining good performance. To that end there is a trend in modern cache design to use larger cache blocks. On uniprocessors their effect is largely positive: large cache blocks exploit spatial locality and consequently reduce first reference misses, by prefetching data and instructions that are adjacent to the address that was referenced. However, on cache-coherent, shared-memory multiprocessors they may have negative effects as well, the most important being an increase in coherence operations from inter-processor accesses to data within the block. Several studies have shown that large cache blocks increase coherence overhead, even to the point where it negates the prefetching benefit [18, 5, 9]. This additional coherence overhead is caused by a phenomenon known as *false sharing* [27, 11].

False sharing is caused by a mismatch between the layout of write-shared data in memory, the cross-processor memory reference pattern to it, and the cache block size. Consider the example shown in Figure 1, in which four processors are accessing different words in the same cache block in a multiprocessor that uses a write-invalidate coherency protocol [1]. Initially, a copy of the cache block is resident in the caches of all four processors. When processor P0 performs a write (to the word marked with an"X"), all copies but its own are invalidated. This causes P3 and P2 to miss when they attempt to reread (different) words in the block. Finally, P1 misses on its write, invalidates the other blocks, and the cycle begins again. All of these operations are due to false sharing – not one is the product of actual (true) data sharing between the processors. However, if the memory layout of the data had been different, so that data accessed by one processor had

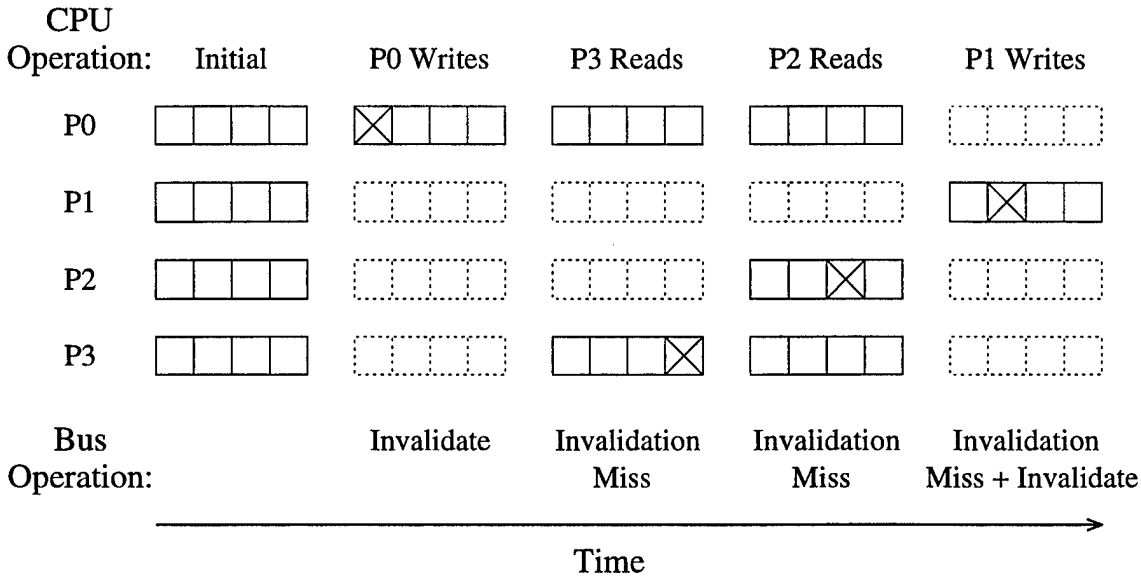

Figure 1: Example of misses caused by false sharing.

not been allocated in the same block with data accessed by the other processors, none would have occurred.

The mismatch between the memory layout of shared data and the processors' access behavior can be eliminated by restructuring the data to match the pattern. In particular, data that is only or overwhelmingly accessed by the same processor can be grouped together, and write shared data objects with no processor locality can be allocated to separate cache lines. The objective of this research was to devise compiler algorithms to restructure shared data in this way, so that false sharing would be eliminated or substantially reduced. Since false sharing is a sizable proportion of memory traffic [28, 11], program execution time should also decrease.

2 Technical Approach

The restructuring can be done either by the applications programmer or the compiler. Manual restructuring, although somewhat successful [27, 11, 25, 24], can be cumbersome, because it requires that the programmer be able pinpoint the data structures that suffer from false sharing in a particular memory (cache) architecture. To identify these data structures, the programmer must know both their layout in memory and the dynamic cross-processor memory reference pattern to them, as well as details of the cache architecture and coherency protocol. All this is hard to determine; knowledge of how each data object is shared is often non-intuitive, and each application must be tailored to the particular cache architecture of the system on which it is running.

For these reasons we chose to automate shared data restructuring via the compiler. We developed a series of compile-time algorithms that analyze coarse-grained, explicitly parallel programs and determine which data structures are susceptible to false sharing, where processor locality may be improved, and whether and how to transform the shared data at compile-time. The algorithms approximate the per-processor memory reference pattern of a program, and determine when it interferes with the layout of shared data in memory. Since false sharing is a product of different

processes accessing different portions of shared data, the analysis focuses on detecting differences in the memory reference pattern between the processes.

The compiler analysis involves three separate stages: (1) per-process control flow analysis, (2) an enhanced, interprocedural, flow-insensitive, summary side-effect analysis [3, 2, 22, 6], and (3) non-concurrency analysis [21] based on barrier synchronization. All analyses were incorporated into the Parafrase-2 [23] source-to-source translator.

(1) Per-process references to shared data may occur as a result of the processes executing different code, and thus accessing different elements of shared data. The per-process control flow analysis [14] determines which sections of code each process executes by computing its control flow graph. In coarse-grained parallel programs, where the parallelism spans very large portions of the programs, the control flow of the processes is neither simple nor regular. Consequently, our algorithm is a general technique that recognizes, separates and groups the processes according to their flow of control over arbitrary control flow graphs.

(2) Per-process references to shared data also occur by the implicit partitioning of arrays across the processes when they execute the same code. The summary side-effect analysis detects this case. To improve its accuracy, we have augmented the traditional analysis with static estimates of execution frequencies [30] and algorithms for merging bounded regular section descriptors [13] lazily and selectively.

(3) The non-concurrency analysis uses barrier synchronization points to determine which portions of a program can execute in parallel and which cannot. It therefore enables the analysis to approximate the memory access pattern of distinct phases of a program (between barriers), and, more importantly, when the pattern shifts. When coupled with static estimates of execution frequencies, it estimates the dominant sharing pattern in the program and restructures for that pattern [15].

Once all stages of the static analysis have been performed, we use a number of heuristics to detect which data structures are susceptible to false sharing and which of three transformations (group and transpose of statically allocated vectors, indirection for dynamically allocated data, and pad and align of both write-busy scalars and locks) should be applied to eliminate it.

2.1 Methodology

In addition to algorithm development, we carefully measured the performance of the restructured programs, comparing them to both unstructured and programmer-restructured versions. Execution times were obtained by executing the programs on a KSR2 [17]; component metrics were derived from shared-memory multiprocessor simulations.

We used a 56 processor KSR2 in which each processor had a 512 KB L1 cache, divided equally between data and instructions. The L2 cache was 32MB and used a coherency unit of 128 bytes. The L2 miss latency was 175 cycles if serviced by a processor on the same ring, and 600 cycles if the servicing processor was on a different ring.

The KSR2 default lock data structure is large (80 bytes) and aligned on cache block boundaries. To make implementation-independent comparisons with the simulations, and to study the effect of padding and aligning locks, we used KSR2 synchronization primitives to provide a smaller (1 word), alternative implementation of locks in the untransformed versions of the programs.

The simulations were trace-driven [12], and traces were obtained for all three versions of each program: unstructured, compiler-restructured and programmer-restructured. The simulator emulates a simple, bus-based shared memory architecture, where the processors are assumed to be

RISClike, with a 32KB L1 data cache and an infinite L2 cache [1]. Several key cache configurations, such as block size, were parameterized, and we studied several settings (4 to 256 byes, in the case of block size). Metrics included different types of miss rates (false sharing, invalidation, shared data and total), bus traffic and total execution cycles.

Gauging the impact of the static algorithms and transformations on program performance and comparing the compiler analysis to programmer efforts to eliminate false sharing required three versions of each program: an unoptimized version, a compiler-restructured version and a programmer-optimized version. The programs we collected had been hand-optimized for locality to varying degrees. In one group, that included Maxflow [4], Pverify [20] and Topopt [7], no effort had been made to improve locality. For Pverify and Topopt, in particular, the programmers had constructed data structures to match their "natural" way of thinking about the semantics of the program algorithms, rather than for better memory system performance. To provide hand-optimized versions of these programs, we manually transformed them [11].

In another group that comprised the original SPLASH benchmark suite [26] (LocusRoute, Mp3d, Pthor, Water) and the SPLASH2 benchmarks (Fmm [25], Radiosity and Raytrace [24]), programs had been highly optimized for locality, including eliminating false sharing. The SPLASH2 programs contained several easily identifiable data structures whose elements had been organized by processor (in our terminology, grouped and transposed), and padded. We undid these transformations to produce unoptimized versions of the programs, but made no other changes. In addition to providing a general comparison between the compiler-directed and hand-tuned optimizations, these hand-unoptimized programs enabled us to gauge the compiler's ability to detect and transform data structures the programmer had chosen. The programmer efforts to improve locality in the original SPLASH benchmarks were not as obvious. Therefore we left them as is.

3 Accomplishments

Applying the compiler algorithms and transformations to the suite of explicitly parallel programs improved program execution in several respects [16].

- False sharing was reduced by an average of 66%, and for some programs over 90%.
- The restructured programs executed up to three times faster than their untransformed counterparts on the KSR2 multiprocessor [17]. Within this range there was some variation. Programs with less speedup had less false sharing to begin with, and therefore, having less of a problem to solve, had less gain. Programs with larger amounts of false sharing reaped more benefit.
- In addition to faster execution times the restructured programs had considerably better scalability (better performance with increasing numbers of processors). In fact, for one program, we ran out of processors (48) before the speedup curve hit a knee.
- When compared to programmer efforts to restructure shared data (including the extensively hand-tuned SPLASH and SPLASH2 benchmarks, our compiler-directed approach never did worse, and for most programs did considerably better.

[1] Infinite caches can be used to approximate very large (on the order of several megabytes) L2 caches [10]

- The performance benefits were obtained with only a 5% increase in compile time (on average).

In all of this, no single algorithm nor single transformation was responsible for the performance improvements; rather, it was their combination that produced these results.

4 Importance of the Accomplishments

The importance of the results (aside from the impressive speedups) is that they make the difficult task of writing efficient parallel programs much easier. Programmers can now write explicitly parallel programs in a manner that naturally expresses the algorithms of the application, without adhering to a "cache conscious" or the even more difficult "cache coherence conscious" programming style. In addition, the ease of programming comes with no cost in parallel program execution (often one must make this tradeoff); on the contrary, the compiler-directed algorithms produced faster programs than those written by programmers who were trying to reduce false sharing and achieve better processor locality manually.

5 Transitions of Research

There are none that I know of, although both the algorithms and restructured parallel programs are available.

6 WEB/ftp Sites with More Information

Much of the software produced to do this work is available on the Web.

- An inline trace generator was developed for generating memory reference traces of parallel programs running on shared-memory multiprocessors [12]. The trace generator was used to generate the traces that were used in performance component analyses of the drawbacks of false sharing and the benefits of compiler-directed shared data restructuring. The source for the generator is available in http://www.cs.washington.edu/research/arch/mptrace.html.
- The restructuring algorithms were incorporated into Parafrase-2 [23] source-to-source translator, which is available from the University of Illinois.
- The source for the restructured programs will soon be available in http://www.cs.washington.edu/homes/eggers/Research/tools.html.
- All papers are available in http://www.cs.washington.edu/homes/eggers/Research/fs.html.
- TorJeremiassen's thesis is available in http://cm.bell-labs.com/who/tor.

7 Additional Remarks

The student that was supported on the project now works at AT&T Bell Laboratories. We are continuing to collaborate on the research. Our current work compares the compiler approach to cache coherence hardware that was designed to do the same. Hardware should be more exact and

therefore produce better performance; however, our software approach is considerably cheaper. The open issue is how close the tradeoffs are.

Simultaneous Multithreading

8 Technical Objectives

The objective of this research was to develop a processor architecture that could provide high processor utilization by addressing its two major impediments: long latency operations and limited per-thread parallelism. The resulting design we call *simultaneous multithreading* (SMT). Simultaneous multithreading permits multiple independent threads to issue instructions each cycle. It combines the multiple-instruction-issue features of modern superscalar processors with the latency-hiding ability of multithreaded architectures. Unlike conventional multithreaded architectures which depend on fast context switching to share processor execution resources, all hardware contexts in an SMT processor are active simultaneously, competing each cycle for all available resources. This dynamic sharing of the functional units allows simultaneous multithreading to substantially increase throughput, attacking both long latencies and limited per-thread parallelism. The ONR grant funded the initial studies that showed simultaneous multithreading's potential.

9 Technical Approach

We first (1) measured the size and nature of the low processor utilization problem on a wide-issue superscalar and then (2) determined the potential of simultaneous multithreading to fix it, comparing it to two alternative architectures designed to address the same issues [29].

(1) The Problem (and Methodology). To motivate the need for simultaneous multithreading, we did a series of experiments that exposed the limits of wide superscaler execution, and bounded the potential improvement possible from latency-hiding techniques that address only one source of the performance loss. Simultaneous multithreading, of course, is a general technique that attacks all sources.

The measurements were based on a workload composed of the SPEC92 benchmark suite [8]. The programs were compiled using the optimizing Multiflow compiler [19]; compiler options were chosen to maximize each program's single-thread performance, in order to give the single-thread executions an advantage when compared to simultaneous multithreading. The programs were then executed on an emulation-based, instruction-level simulator, which simulates a superscalar architecture based on the 300MHz DEC Alpha 21164, extended for wider superscaler execution, better branch prediction and larger, higher-bandwidth caches.

(2) SMT Potential. To demonstrate the potential advantage of simultaneous multithreading, we defined and analyzed several possible SMT machine models and compared them to alternative architectures that attempt to solve the low processor utilization problem. The SMT models span a range of hardware complexities and reflect several possible design choices that differ in how threads can use issue slots and functional units each cycle. For all models the underlying machine was the wide superscalar used above: 10 functional units capable of issuing 8 instructions per cycle.

10 Accomplishments

The results [29] demonstrated that the functional units of a wide superscalar processor are highly underutilized. The average utilization across the entire workload was only 19%, which represents

an average execution of less than 1.5 instructions per cycle.

They also indicated that there was no dominant source of wasted issue bandwidth. Although there were dominant items in individual applications (e.g., short floating point latencies or data cache misses), they were different in each case. On average, the largest cause (short FP dependences) was responsible for 37% of the issue bandwidth, but there were four other causes that accounted for at least 5.4% of wasted cycles each. Completely eliminating any one factor would not necessarily improve performance by a comparable amount, because many of the causes overlapped.

Because there was no dominant cause of wasted cycles, it is unlikely that a latency-tolerating technique that attacks a specific type of latency will provide a solution. Dramatic increases in performance will only come from a general latency-hiding technique, one that can eliminate multiple causes of latency. Simultaneous multithreading is such a technique.

The experiments that evaluated the simultaneous multithreading models showed maximum speedups over wide-issue superscalars ranging from 3.2 to 4.2, with an issue rate as high as 6.3 instructions per cycle. Even simplified implementations of simultaneous multithreading with limited per-thread capabilities achieved higher instruction throughput than the superscalar.

The results also showed that one of the alternatives to simultaneous multithreading, fine-grain multithreading, achieves only half the performance of an SMT processor: a maximum speedup of only 2.1 over wide-issue superscalar execution (3.2 instructions per cycle). Fine-grain multithreading architectures utilize some unused cycles by executing instructions from a different thread; however, unlike simultaneous multithreading, in any given cycle they are restricted to executing instructions from only one thread.

Simultaneous multithreading also outperformed the other alternative, single-chip multiprocessors, in a variety of configurations. This is again due to the dynamic way in which SMT resources can be scheduled among threads; multiprocessors have a static partitioning of resources to threads, which limits their performance. For example, a single 8-thread, 8-issue SMT processor with 10 functional units was 24% faster than an 8-processor, single-issue MP, which had identical issue bandwidth but required 32 functional units; to equal the throughput of the 8-thread 8-issue SMT, an MP system required eight 4-issue processors, which consumed 32 functional units and 32 issue slots per cycle.

11 Importance of the Accomplishments

The importance of the results described above is that they show that a simultaneous multithreading processor has the potential to solve *two* current stumbling blocks to better processor performance: long latencies and limited per-thread parallelism. Since both factors have multiple causes, a general technique that attacks all sources of lost cycles is required, and simultaneous multithreading is such a technique. The experiments also indicated that simultaneous multithreading should outperform alternative architectures that address these same sources of performance loss: wide-issue superscalars, coarse-grained and fine-grained multithreaded architectures and single-chip multiprocessors.

12 Transitions of Research

Our industrial partner in this work is Digital Equipment, Corp. We are working with them on all aspects of the project: Digital provided source code for our compiler platform (Multiflow); DEC processor designers and researchers consulted on the architectural design; the architects actively participated in the research; and we transfer all design and software technology to them. One could imagine that they are designing a production SMT processor.

The research on simultaneous multithreading has just begun. We expect that, as the work progresses, we will interface with other companies. For example, we have recently given talks on the processor architecture at Hewlett Packard, Intel, and Silicon Graphics.

13 WEB/ftp Sites with More Information

A copy of our first paper (the ONR-sponsored work) can be found in
http://www.cs.washinton.edu/research/smt, the 1995 paper. Other papers in
http://www.cs.washinton.edu/research/smt reflect work that has been accomplished since the original ONR seed funding.

14 Additional Remarks

We have a grant from NSF's Experimental Systems program to continue the research in simultaneous multithreading. The objective of our future research is to explore the many issues and tradeoffs in the design of a simultaneous multithreaded processor, so that we can develop a low-level architectural specification that will lead to eventual hardware design. We will focus in particular on multi-thread instruction fetching and scheduling, register file, functional unit and pipeline design, and speculative execution hardware. We will also develop compiler and operating systems support for the creation, optimization, and synchronization of threads executing on a simultaneous multithreaded processor. For this work we will target both multi-programmed, and medium and coarse-grained parallel workloads.

The first student supported on the project has taken a faculty postition at the University of California, San Diego. We are continuing to collaborate on the research.

15 Bibliography

References

[1] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4:273–298, November 1986.

[2] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Symposium on Principles of Programming Languages*, January 1979.

[3] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9), September 1978.

[4] F.J. Carrasco. A parallel maxflow implementation. Technical report, March 1988.

[5] D.F. Cheriton, A. Gupta, P.D. Boyle, and H.A. Goosen. The vmp multiprocessor: Initial experience, refinements and performance evaluation. In *15th Annual International Symposium on Computer Architecture*, May 1988.

[6] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Conference on Programming Language Design and Implementation*, June 1988.

[7] S. Devadas and A.R. Newton. Topological optimization of multiple level array logic. *IEEE Transactions on Computer-Aided Design*, November 1987.

[8] K.M. Dixit. New CPU benchmark suites from SPEC. In *COMPCON*, February 1992.

[9] S.J. Eggers. Simulation analysis of data sharing in shared memory multiprocessors. Technical Report No. UCB/CSD 89/501 (Ph.D. thesis), University of California, Berkeley, March 1989.

[10] S.J. Eggers. Simplicity versus accuracy in a model of cache coherency overhead. *IEEE Transactions on Computers*, 40(8), August 1991.

[11] S.J. Eggers and T.E. Jeremiassen. Eliminating false sharing. In *International Conference on Parallel Processing*, volume I, August 1991.

[12] S.J. Eggers, D. Keppel, E. Koldinger, and H.M. Levy. Techniques for inline tracing on a shared-memory multiprocessor. In *1990 ACM Sigmetrics*, volume 18, May 1990.

[13] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *ACM Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[14] T.E. Jeremiassen and S.J. Eggers. Computing per-process summary side-effect information. In *5th Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[15] T.E. Jeremiassen and S.J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *International Conference on Parallel Architecture and Compilation Techniques*, August 1994.

[16] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Symposium on Principals & Practice of Parallel Programming*, July 1995.

[17] Kendall Square Research. *KSR/Series Principles of Operations*, revision 7.0 edition, 1994.

[18] R.L. Lee, P.C. Yew, and D.H. Lawrie. Multiprocessor cache design considerations. In *14th Annual International Symposium on Computer Architecture*, June 1987.

[19] P.F. Lowney, S.M. Freudenberger, R.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7, 1993.

[20] H-K. T. Ma, S. Devadas, R. Wei, and A. SangiovanniVincentelli. Logic verification algorithms and their parallel implementation. In *Proceedings of the 24th Design Automation Conference*, July 1987.

[21] S.P. Masticola and B.G. Ryder. Non-concurrency analysis. In *Symposium on Principals & Practice of Parallel Programming*, May 1993.

[22] E. Myers. A precise inter-procedural data flow algorithm. In *Symposium on Principles of Programming Languages*, January 1981.

[23] C. Polychronopoulos, M. Girkar, M. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Paraphrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *International Conference on Parallel Processing*, volume II, August 1989.

[24] J.P. Singh, A. Gupta, and M. Levoy. Visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7), July 1994.

[25] J.P. Singh, C. Holt, J.L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Supercomputing 93*, November 1993.

[26] J.P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[27] J. Torrellas, M.S. Lam, and J.L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *International Conference on Parallel Processing*, volume II, August 1990.

[28] J. Torrellas, M.S. Lam, and J.L. Hennessy. False sharing and spatial locality in multiprocessor caches. *toc*, 43(6), June 1994.

[29] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[30] D.W. Wall. Predicting program behavior using real or esitmated profiles. In *Conference on Programming Language Design and Implementation*, June 1991.